Modeling Out-of-order Processors for Software Timing Analysis

Xianfeng Li       Abhik Roychoudhury       Tulika Mitra
National University of Singapore

WCET Analysis
- Program path analysis [Shaw’89, Healy’98...]
  - All paths in control flow graph are not feasible
- Micro-architectural modeling
  - Dynamically variable instruction execution time
    - Cache, Pipeline [Li’99, Schneider’99, Thieling’00...]
    - Branch Prediction [Colin’00, Mitra’02]
    - Out-of-order pipelines [Heckmann’03]
      - Based on Abstract Interpretation
- We propose a new method for modeling out-of-order executions

Complications
- Several instructions may reside in the same pipeline stage in the same clock cycle
  - An ADD instruction and MUL instruction in the EX stage since they use different functional units
- Pipeline stalls
  - Instruction I+1 may not proceed to EX since it depends on the result of instruction I
- Mask stall latency by out-of-order exec
  - If I+1 cannot proceed, let I+2 proceed if all its operands are available

Worst Case Execution Time (WCET)
- Maximum execution time of a program on a micro-architecture for all possible inputs
  - Required for schedulability analysis
- Measurement for all inputs: impractical
  - Execute program for selected inputs to get a lower bound on WCET (Observed WCET)
- Analysis
  - Employ static analysis to compute an upper bound on WCET (Estimated WCET)

Pipelined program execution
- Divide the execution of an instruction into stages.
  - Instruction I+1 can proceed before I completes.
  - Increased throughput, lower overall execution time.

An out-of-order pipeline
- Taken from SimpleScalar architecture simulator
  - Mem => I buffer (in-order)
  - IBUF => ROB (in-order)
  - ROB => FU (out-of-order), (Instr still in ROB)
  - FU => ROB (out-of-order) (forward data)
  - Update register file, free ROB entry (in-order)
**Sample out-of-order exec.**

### Instruction sequence
- **A**: `mult r3 r1 r2`
- **B**: `add r3 r3 8`
- **C**: `and r3 r3 0xff`
- **D**: `addu r5 r4 8`
- **E**: `mult r5 r5 r6`

### Latencies
- **MULTU**: 1 ~ 4 cycles
- **ALU**: 1 cycle

### Partial order of dependences
- A → B → C → D → E

**Instruction A executes 3 cycles**

**Instruction A executes 4 cycles**

**Timing Anomaly**

- Overall WCET of an instruction sequence cannot be obtained from WCET of each instruction
  - Need to consider all possible execution times of each instruction to safely estimate WCET!
    - Expensive enumeration
  - Very different from cache modeling
    - Worst-case cache behavior of an instruction sequence can be safely estimated by considering all cache accesses as misses

**To-do list for pipeline modeling**

- Safe estimation of WCET of an instruction sequence
  - Must avoid enumerating all instruction schedules
- Tight estimation of WCET of an instruction sequence
  - Must avoid too much pessimism in estimating the contentions among instructions
- Estimating WCET of whole program
  - Estimating straight-line code starting from different pipeline states
  - Beyond straight-line code

**Abstract interpretation approach**

- Exec. of an instruction in an abstract pipeline state
  - **IF**: Hit or miss in instruction cache
    - 1 cycle
  - **ID**: Instruction Decode
    - 1 cycle or more (if ROB is not free)
  - **EX**: Variable Latency instruction
    - 1 ~ 5 cycles
  - **WB**: Forward results to ROB
    - 1 cycle
  - **CM**: Wait for previous instructions to commit
    - 1 ~ 4 cycles (if ROB holds 4 instructions)
  - How to deal with such non-determinacy of timing?
Find WCET of B

Model execution of B as an Execution Graph

Basic block B in control flow graph of program
- Assume clean pipeline state before B

Model execution of B as an Execution Graph
- Nodes of the graph are IF(I), ID(I) etc
- Dependence edges: x → y
  - x must finish before y starts in any schedule
- Contention edges: x → y
  - x may start before y starts, delaying y

Find WCET of B
- Compute interval for ready/start/end of each node.

Abstract interpretation approach
- Consider all possibilities,
  - IF may result in hit or miss
  - EX may take 1 or 2 or 3 or 4 or 5 cycles
- ... but exclude those which can be ruled out via AI over suitable domains
  - IF of instruction I will never result in a cache miss
  - EX of instruction I will always take 1 cycle
- Propagate the successor states to find all possible pipeline states at a control location (fixed point)
  - WCET of a single instruction [Heckmann03]
  - Whole program’s WCET by Integer Linear Programming

Avoid enumeration, locally first!
- EX of I may take 1 or 2 or 3 or 4 or 5 cycles
  - EX of I takes [1,5] clock cycles
  - But ...
    - Depending on how many clock cycles EX takes, the contentions of the to-be-exec instructions are decided
    - the future pipeline state is decided by such timing
- How do we calculate possible pipeline states at each control location?
  - Cannot case split based on the possible timings of EX(I), of course
  - Can we avoid this altogether?

Use Intervals uniformly
- Ready/Start/End of each pipeline stage of each instr.
  - IF(I), ID(I), EX(I), WB(I), CM(I)
  - Bound by an interval
  - WCET of I = latest_{EX(I)} - earliest_{IF(I)}
- Interval for EX(I) includes
  - Time to execute I
  - Time to wait for exec I: resolving data dependences
  - Time to wait for executing I: contention for FU from
    - Instructions earlier than I in program
    - Instructions that may I in program
  - Must consider all possibilities without enumeration

Computing Intervals – Basic Idea
- Consider all pairs (I, J) ∈ Instr × Instr
  - Assume all instr. J can delay all instr. I
  - Leads to very coarse intervals for
    - IF(I), ID(I), EX(I), WB(I), CM(I)
    - hence very coarse WCET of I
  - Rule out certain contentions
    - Data dependencies, for starters!
    - earliest_{IF(I)} ≤ latest_{IF(J)}
    - earliest_{IF(I)} ≤ latest_{IF(J)}
  - Refine to get tighter estimates
    - This again rules out some more contentions!
  - ... until you reach a fixed point

Formal treatment

Execution Graph

<table>
<thead>
<tr>
<th>cycle</th>
<th>end,IF</th>
<th>end,EX</th>
<th>end,WB</th>
<th>end,CM</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,5]</td>
<td>[0,3]</td>
<td>[0,3]</td>
<td>[0,3]</td>
<td>[0,3]</td>
</tr>
<tr>
<td>[1,5]</td>
<td>[1,3]</td>
<td>[1,3]</td>
<td>[1,3]</td>
<td>[1,3]</td>
</tr>
<tr>
<td>[2,5]</td>
<td>[2,3]</td>
<td>[2,3]</td>
<td>[2,3]</td>
<td>[2,3]</td>
</tr>
<tr>
<td>[3,5]</td>
<td>[3,3]</td>
<td>[3,3]</td>
<td>[3,3]</td>
<td>[3,3]</td>
</tr>
<tr>
<td>[4,5]</td>
<td>[4,3]</td>
<td>[4,3]</td>
<td>[4,3]</td>
<td>[4,3]</td>
</tr>
<tr>
<td>[5,5]</td>
<td>[5,3]</td>
<td>[5,3]</td>
<td>[5,3]</td>
<td>[5,3]</td>
</tr>
</tbody>
</table>
Schedulability Analysis

- Processes in task graph allocated to Processors
  - Dependency among processes (edges of task graph)
  - Contention among processes (based on allocation)
  - Priorities for processes allocated to same Processor
  - Calculate Worst case completion time [Yen & Wolf 98]

- Exec. graph: Dependency/contention between nodes
  - Priorities determined by program order of instructions
  - Cannot use the result directly, e.g.
    - If I+k is executing when I becomes ready, I+k delays I
    - Later (lower priority) instructions affect worst case completion of EX(I)

To-do list

- Safe estimation of WCET of an instruction sequence
  - Must avoid enumerating all instruction schedules
- Tight estimation of WCET of an instruction sequence
  - Must avoid too much pessimism in estimating the instruction contentions
- Estimating WCET of whole program
  - Estimating straight-line code starting from different pipeline states
  - Beyond straight-line code

Bounding contexts

- For a basic block B
  - Instructions before B which directly affect the exec. of B -- Prologue
  - Similarly, Epilogue
  - Size of Prologue and Epilogue decided by architectural parameters e.g. ROB

- Only dependence/contention from prologue/epilogue considered for estimating WCET(B)
  - Requires estimating intervals for IF/ID of instr. in prologue and epilogue
  - Done conservatively by assuming max. contention
  - Imposes finite bound on the context for B

Handling sequences of Basic Blocks

IF(1) can happen much before CM(0) - pipeline/o-o-exec
Overlap = End of CM(0) - Ready of IF(1)
Calculate minimum overlap by analyzing exec. graph.
WCET(B2 with B1 as prologue) =
latestend, CM(N) - earliest end, CM(0)

Estimate for entire program

- Define a constant wcet(B) for each basic block B
  - Max over WCET(B) with all possible prologue/epilogue
- WCET of a program T (maximize via ILP solver)
  - T = ∑ B wcet(B) * n_B
  - n_B is a variable denoting number of executions of B
  - Bound n_B via inflow-outflow constraints, loop bounds

Other architectural features

- Instruction Cache
  - Classify instructions (similar to AI approach)
  - Always Hit, Always Miss, First Miss, Unknown
  - Modify execution time of IF(I) based on classification
    - 1 cycle without cache modeling
    - N cycle if I is classified as always miss
    - Approximated by [1,N] if I is classified unknown
    - Shows flexibility of the interval based modeling
- Branch prediction
  - Involves changes to Execution Graph
  - Extremely involved, see Technical Report for full modeling and detailed proofs
Experimental Results -- Pipeline + I-Cache

### Parameters:
- **Func. Units:** ALU: 1 cycle; MUL: [1, 4]; FPU: [1, 12]
- **4KB I-Cache:** 4-way, 32 sets, 32 bytes/line, cache miss: 10 cycles

<table>
<thead>
<tr>
<th>Program</th>
<th>Obs. WCET</th>
<th>Est. WCET</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>matsum</td>
<td>100597</td>
<td>111193</td>
<td>1.10</td>
</tr>
<tr>
<td>test</td>
<td>100654</td>
<td>12240</td>
<td>1.15</td>
</tr>
<tr>
<td>fft</td>
<td>1005416</td>
<td>1270086</td>
<td>1.17</td>
</tr>
<tr>
<td>what</td>
<td>909531</td>
<td>1029380</td>
<td>1.13</td>
</tr>
<tr>
<td>fiv</td>
<td>44310</td>
<td>50426</td>
<td>1.15</td>
</tr>
<tr>
<td>ludcomp</td>
<td>11946</td>
<td>16283</td>
<td>1.36</td>
</tr>
<tr>
<td>mixer</td>
<td>8235</td>
<td>11062</td>
<td>1.34</td>
</tr>
</tbody>
</table>

Experimental Results -- Pipeline + Branch Prediction

### Parameters:
- **Func. Units:** ALU: 1 cycle; MUL: [1, 4]; FPU: [1, 12]
- **Gag dynamic branch predictor:** 4-bit BHR, 16-entry BHT

<table>
<thead>
<tr>
<th>Program</th>
<th>Obs. WCET</th>
<th>Est. WCET</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>matsum</td>
<td>101628</td>
<td>111744</td>
<td>1.10</td>
</tr>
<tr>
<td>fest</td>
<td>9038</td>
<td>10535</td>
<td>1.15</td>
</tr>
<tr>
<td>fft</td>
<td>1005406</td>
<td>1285063</td>
<td>1.17</td>
</tr>
<tr>
<td>whet</td>
<td>913296</td>
<td>1011276</td>
<td>1.14</td>
</tr>
<tr>
<td>fir</td>
<td>45867</td>
<td>50159</td>
<td>1.17</td>
</tr>
<tr>
<td>ludcomp</td>
<td>11157</td>
<td>14325</td>
<td>1.28</td>
</tr>
<tr>
<td>mixer</td>
<td>7053</td>
<td>9222</td>
<td>1.28</td>
</tr>
</tbody>
</table>

Summary

- Out-of-order pipelined execution involves a complicated instruction scheduling.
  - Timing of the instruction scheduling depends on
    - Dependence between instructions (data hazard)
    - Contention between instructions (resource hazard)
- We use schedulability analysis methods for tasks with dependence and contention [Yen & Wolf 98]
  - Avoid enumeration of cases with interval based modeling of pipeline evolution
  - Fixed point on intervals unlike AI approach.
  - Integrated with other micro-architectural features.
  - Currently working on expt on a processor with out-of-order pipeline, instruction cache and branch prediction